商用多核处理器中的存储器一致性模型与高速缓存同一性协议

周君 唐士斌

1 引言

当前，人类对各类计算技术的需求越来越多。电子及微电子技术作为其中的排头兵，受到学术界和工业界高度关注。在单芯片上集成更多数量的晶体管，可以在单芯片上获得更复杂的功能和更高的性能。然而，由于受到芯片中信号传播速度和加工工艺的限制，依赖于高频率的单处理器芯片的运算性能难以进一步提升。为此，充分挖掘程序中指令级并行的超标量技术在处理器设计中得到广泛使用。然而，超标量处理器的资源利用率不高，并会随着指令发射宽度的增加进一步加剧片上资源的浪费；另一方面，超标量结构的设计与验证复杂度随着设计规模的增长呈超线性趋势上升。因此，在超标量技术提供技术进步的能力难以为继的情况下，为了有效利用片上的资源，多核结构作为一种挖掘处理器并行性能的技术，逐步占据了当前微处理器设计领域的主导地位。相对于超标量处理器，多核处理器在性能上有50%-100%的提升[1]。

多处理器系统存在多种组织方式，可以分为共享存储结构和非共享存储结构[2]，前者对系统内存进行统一编址，进程可以直接访问本地节点和其他节点主存中的数据。非共享存储结构也可以共享主存，但是对于非本地节点的主存不能直接访问，需通过软件辅助间接访问，时间开销较大。共享存储的多处理器系统还可以分为集中式共享存储结构和分布式共享存储结构[3]。其中集中式共享存储结构又称为均匀访存延迟的存储结构（UMA），又可分为并行向量处理机（PVP）和对称多处理器系统（SMP）。分布式共享存储结构包括非均匀访存延迟的存储结构（NUMA）和无远程访问结构（NORMA）。

在共享存储的多处理器系统中，保障存储器一致性（Consistency）和高速缓存（Cache）同一性（Coherency）是两大关键技术。前者规定了访问其他存储器地址的时间关系，是用户和系统之间的合同或者协议，用以确保多个处理器看到的存储器视图是一致的；后者定义了访问同一存储器的空间关系，目的是使主存中数据的备份保持最新。二者在多处理器系统中举足轻重，不仅直接决定系统的正确性，并且对系统的规模和性能有至关重要的影响。

本文将就多核处理器中的存储器一致性模型和高速缓存同一性协议分别作简要描述，并重点介绍与分析当前几种典型的商用处理器在以上两方面的技术进展情况。

2 存储器一致性

2.1 存储器一致性的基本概念

共享存储系统中的存储器一致性模型是一份软件程序员与系统实现的契约，该契约规定了系统执行的访存操作如何展现给程序员。程序编写过程中，为了得到预期的结果，程序员需要考虑存储器一致性模型对访存操作顺序的影响。因此，存储器一致性模型影响了可编程性。此外，存储器一致性模型对访存操作顺序的限制约束了底层的具体实现（例如：是否允许乱序？是否保证写原子性等等），因此，存储器一致性模型同时也影响了系统的性能。存

储器一致性模型贯穿于并行程序编写与系统设计的每一个细节（包括：处理器、互连网络、编译器和编程语言）当中。

在实际实现中，存储器一致性模型决定了程序在共享存储系统中执行时读操作的返回值。直观地说，读操作应该返回对该地址最近一次写的值。在单处理器系统中，该条件很容易满足，“最近一次”受“程序序”的约束（也就是，访存操作在程序中出现的顺序）。但在多处理器系统中，情况要复杂得多。

下面，我们重点介绍几种商业处理器中采用的存储器一致性模型。

2.2 商业处理器中的存储器一致性模型

2.2.1 SPARC 处理器的一致性模型[13]

2.2.1.1 基本概念

本节我们首先介绍一下 SPARC 处理器存储一致模型过程涉及的一些基本概念：

1. 访存事务

访存事务可以分为以下四类：

1) **写操作** 是由处理器发起的写请求。当写操作的值对系统中所有处理器可见时表示写操作完成。
2) **读操作** 是由处理器发起的读请求。当读操作的值返回并且不能被其他任何处理器的写修改时表示读操作完成。
3) **原子操作** 是一组读改写的操作。系统需要保证在读写原子操作之间没有其他的访存事务影响内存的状态。SPARC-V9 中的原子指令包括：LDSTUB、SWAP、CAS。
4) **刷新操作** 是由处理器发出的请求。用来保证数据空间与指令空间的一致。

同时需要注意的是： MEMBAR 指令并非访存事务。

2. 程序序

程序序是每个处理器的一个全序关系，该序表示处理器逻辑上执行指令的顺序。MEMBAR 指令可以用来约束程序序。

3. 依赖序

指令依赖序是一种偏序关系，该关系表示同一处理器的两条指令访问相同的寄存器或者内存地址时的顺序。

4. 存储序

在内存端执行的访存事务的顺序，所有的访存事务具有全局的存储序。

5. MEMBAR 指令

MEMBAR 指令可以用来约束单处理机指令流的序。MEMBAR 之前的读写操作在程序序上优先于 MEMBAR 之后的读写操作。此外，原子操作也受到 MEMBAR 指令的约束。MEMBAR 指令的分类如表 1 所示。

表1. MEMBAR 指令

MEMBAR 汇编指令	对存储序(<m)的约束
MEMBAR #LoadLoad	Load <m Load
MEMBAR #StoreLoad	Store <m Load
MEMBAR #LoadStore	Load <m Store
MEMBAR #StoreStore	Store <m Store

2.2.1.2 RMO[1]存储一致模型

1. 值的原子性

高速缓存同一性的粒度是以 8 字节为单位，8 字节及以下的单个内存事务一定是原子执行的。

2. 写原子性

写操作具有全局序。然而并不表示该序可见，同时也不表示 SPARC 存储一致性需要中心控制机制。

3. 原子访存事务

原子访存事务（如：SWAP、 LDSTUB 与 CAS）中的读操作与写操作原子执行，任何的访存事务不能将其隔断。

4. 存储序的约束

   1) 当指令与更早的读操作存在依赖序时，RMO 存储一致保证他们之间有序执行。然而，与写操作之间的数据依赖可能因为写操作的完成被推迟，他们之间的顺序不能保证全局一致（本地一定是按序的）。
   2) 用 MEMBAR 指令约束访存指令之间的序。
   3) 同一处理器对相同地址的写操作保证程序序。

5. 访存事务的值

`Store` 操作的值由执行的处理器决定。读操作的值，取决于对该地址最近一次的写。最近一次，具体是指存储序上对该地址最后一次写操作写入的值，抑或是本地处理器写缓存中的写操作的值。

2.2.1.3 PSO[2]存储一致模型

PSO 存储一致模型是在 RMO 存储一致模型的基础上增加了对访存操作的约束，该约束等效于在读操作之后，增加两条隐式的同步指令（MEMBAR `#LoadLoad` | `#LoadStore`）。换句话说，PSO 存储一致模型中增加约束不允许以下两种操作：(1) 读操作超越更早的读操作；(2) 写操作超越更早的读操作。

2.2.1.4 TSO[3]存储一致模型

TSO 存储一致模型是在 PSO 存储一致模型的基础上再次增加对访存操作的约束，该约束等效于在写操作之后，增加两条隐式同步指令（MEMBAR `#StoreStore`）。换句话说，TSO 存储一致模型增加了对写与写之间的约束，不允许写操作超越更早的写操作。

2.2.2 Intel 处理器的一致性[9]

2.2.2.1 2001 年 x86 体系结构的存储器一致性模型

根据 2001 年英特尔发布的软件开发者手册第三卷——系统编程指导，单处理器系统中，对序关系有以下约束：

1) 读操作允许以任意顺序投机执行。

---

[1] Relaxed memory order，放松的访存序
[2] Partial Store Order，写偏序
[3] Total Store Order，写全局序

2) 在写缓存中的写操作彻底完成之前，后续的读操作可以提前执行。
3) 对内存的写操作都是遵照程序序执行的，除了以下两种情况：(1) CLFLUSH 指令引起的写操作、(2) 非暂时的写操作（例如指令：MOVNTI、MOVNTQ、MOVNTDQ、MOVNTPS、MOVNTPD）。
4) 写操作可以缓存在写缓存中。
5) 写操作不能投机执行，写操作必须真正的提交以后才能被实施（例如：写入写缓存）。
6) 处理器发出的读操作可以提前读取同一处理器写缓存中的数据。
7) 普通的访存操作不能超越读写（I/O）指令、具有锁前缀的指令、顺序指令。
8) 读操作不能超越 LFENCE 与 MFENCE 指令。
9) 写操作不能超越 SFENCE 指令。

而多处理器系统中的约束原则如下：

1) 单独的处理器遵守单处理器系统的所有约束。
2) 来自同一处理器的所有写操作，被所有处理器看到的顺序是一致的。
3) 在总线上执行的写操作全局可见，但是来自不同处理器的写操作之间没有默认顺序约定。

2.2.2.2 2008 年 x86 体系结构的存储器一致性模型

根据 2008 年英特尔发布的软件开发者手册第三卷——系统编程指导，单处理器系统中，对序关系有以下约束：

1) 读操作不能与更早的读操作乱序。.
2) 写操作不能与更早的读操作乱序。
3) 对内存的写操作都是遵照程序序执行的，除了以下两种情况：(1) CLFLUSH 指令引起的写操作、(2) 非暂时的写操作（例如指令：MOVNTI、MOVNTQ、MOVNTDQ、MOVNTPS、MOVNTPD）。
4) 读操作在与更早的写操作之间没有数据依赖时，读操作可以超越写操作。
5) 访存操作不能超越读写指令、具有锁前缀的指令、顺序指令。
6) 读操作不能超越 LFENCE 与 MFENCE 指令。
7) 写操作不能超越 SFENCE 与 MFENCE 指令。

多处理器系统中的约束原则如下：

1) 单独的处理器遵守单处理器系统的所有约束。
2) 来自同一处理器的所有写操作，被所有处理器看到的顺序是一致的。
3) 不同处理器之间的写操作没有默认顺序约定。
4) 访存操作之间遵守因果序。
5) 对同一地址的写操作具有全局序。
6) 带有锁前缀的指令具有全局序。

2.2.2.3 2010 年 x86 体系结构的存储器一致性模型

根据 2010 年英特尔发布的软件开发者手册第三卷——系统编程指导，单处理器系统中，对序关系有以下约束：

1) 读操作不能与更早的读操作乱序。.
2) 写操作不能与更早的读操作乱序。
3) 对内存的写操作不能超越更早写操作，除了以下三种情况：

- CLFLUSH 指令引起的写操作
- 非暂时的写操作（例如指令：MOVNTI、MOVNTQ、MOVNTDQ、MOVNTPS、MOVNTPD）。
- String 操作（参考上述手册 8.2.4.1 小节）

4) 读操作在与更早的写操作之间没有数据依赖时，读操作可以超越写操作。
5) 访存操作不能超越读写指令、具有锁前缀的指令、顺序指令。
6) 读操作不能超越 LFENCE 与 MFENCE 指令。
7) 写操作不能超越 LFENCE、SFENCE 与 MFENCE 指令。
8) LFENCE 指令不能超越更早的读操作。
9) SFENCE 指令不能超越更早的写操作。
10) MFENCE 指令不能超远更早的读写操作。

多处理器系统中的约束原则如下：

1) 单独的处理器遵守单处理器系统的所有约束。
2) 来自同一处理器的所有写操作，被所有处理器看到的顺序是一致的。
3) 不同处理器之间的写操作没有默认顺序约定。
4) 访存操作之间遵守因果序。
5) 来自不同处理器的两个写操作以相同的顺序被其它处理器看到，然而执行该指令的处理器允许看到不同的顺序。
6) 带有锁前缀的指令具有全局序。

2.2.2.4 对 x86 体系结构存储一致模型的演变分析

通过上述内容我们发现，x86 体系结构存储一致模型的演变主要集中在了写原子性上，也就是说，写操作是否具有全局序。

表2. X86 软件开发手册对写原子性的描述

2001 年	6) 处理器发出的读操作可以提前读取同一处理器写缓存中的数据。 3) 在总线上执行的写操作全局可见，但是来自不同处理器的写操作之间没有默认顺序约定。
2008 年	4) 访存操作之间遵守因果序。 5) 对同一地址的写操作具有全局序。
2010 年	4) 访存操作之间遵守因果序。 5) 来自不同处理器的两个写操作以相同的顺序被其它处理器看到，然而执行该指令的处理器允许看到不同的顺序。如四个处理器 A、B、C、D，处理器 A 与 B 分别执行 W1 与 W2，处理器 C 与 D 必须以相同的顺序看到 W1 与 W2（W1 优先于 W2，或者相反），然而，允许处理器 A 与 B 以不同的顺序看到 W1 与 W2 的执行。

在对比 x86 体系结构的写原子性之前，首先我们对比三个概念：写原子性、强写原子性、弱写原子性。

1) 写原子性，是指写操作原子完成，隐含了所有处理器的写操作具有全局序（它保证一个处理器在该处理器读到写操作的新值以后及在该值对所有处理器可见之前均不能执行访存操作）；
2) 强写原子性，存储一致模型中默认的写原子性是指强的写原子性；

3) 弱写原子性，在不影响正确性的前提下，人们规定，在写操作全局可见之前，执行写操作的处理器读取了新值以后，可以继续执行访存操作，我们称之为弱写原子性。

通过对比如表格 2 所述的以上三个 x86 软件开发手册中对写原子性的描述，我们发现：

1) 2001 年的描述中明确表达出了 X86 体系结构的存储一致模型具有弱写原子性的特征，具体表现为，同一处理器的读操作可以提前读取写操作的值，在到达总线以后，可以保证写操作的原子执行。

表3. 写原子性举例

初始状态	X=0, Y=0			
同期执行	P1	P2	P3	P4
	X= 1	Y =1		
			R1= X	R3= Y
			R2= Y	R4= X
写原子性禁止的结果	(R1, R2, R3, R4) = (1, 0, 1, 0)			

2) 2008 年的描述中，仅保证了高速缓存同一性（也就是，对相同地址的写操作保证全局序）。并不保证写原子性。如表格 3 所示，在保证写原子性且读操作不能乱序的处理器中，结果(R1, R2, R3, R4) = (1, 0, 1, 0)是被禁止的。该年的描述中，禁止读操作乱序，但是允许该类结果的出现。

3) 2010 年的描述，再次保证了弱的写原子性。具体表现为，对其他处理来说写操作具有全局序，而执行写操作的处理器可以提前读取写操作的值。

因此，综上所述，我们发现 2010 年及其以后的 X86 体系结构存储一致模型，具有弱的写原子性，具有与 SPARC 中 TSO 存储一致模型极为相似的语义[6][7][8] 。

2.2.3 Power 处理器的一致性

在 Power 体系结构的手册[10][11]中，对 Power 存储一致模型的描述概括性较差。为了便于理解，本文不使用官方手册的说明而采用丹尼尔. 索林（Daniel J. Sorin）等对 Power 存储一致模型的总结[12]。

索林等首先定义了一种基本的放松存储一致模型（简称 XC）。该存储一致模型满足以下约束：

1) 相同地址的读写操作之间满足程序序。
2) 在写操作彻底完成之前，同一处理器后续的读操作允许提前读取本地写操作的值。
3) 不同地址的读写操作可以乱序，原子的读改写指令与访问不同地址的读或写指令可以乱序。
4) FENCE 指令不允许与读写操作乱序。
5) 所有处理器的写操作具有全局序。

Power 存储一致模型与基本的放松存储一致模型相比，有以下特点：

1) Store 是相对于其他处理器而非相对于内存而言（也就是说 Power 的写操作没有全局序，不保证写原子性）。例如：处理器 C1 对地址 A 进行的写操作 S1 相对于处理器 C2 执行完成（C2 不能读到 S1 写之前的值），但处理器 C3 依旧能读到 S1 执行之前的旧值。

2) FENCE 操作可以累加。假设处理器 C2 执行访存操作 X1、X2、X3….（用集合 X

表示），然后执行 FENCE 指令，再执行访存操作 Y1、Y2、Y3（用集合 Y 表示）。Power 对累加做如下定义：(1) 其他处理器（如 C1）增加到集合 X 的操作，优先于 C2 的 FENCE 指令执行；(2) 其他处理器（如 C2）增加到集合 Y 的操作，被约束到 FENCE 指令之后执行；(3) 条件(1)可以累加，优先于处理器 C1 的一定优先于 C2 的 FENCE 指令，同理条件(2)也可以累加。

3) Power 存储一致模型提供了三类 FENCE 指令：

- HWSYNC 指令（全称：heavy weight synchronization，重量级同步），集合 X 中所有的操作优先于集合 Y 的所有操作，允许累加。
- LWSYNC 指令（全称：light weight synchronization，轻量级同步），LWSYNC 在 HWSYNC 的基础上，放松了对集合 X 中的写操作与集合 Y 中的读操作的约束，同样允许累加。
- ISYNC 指令（全称：Instruction synchronization，指令同步），约束来自同一处理器的两条读操作，不允许累加，同时该指令仅约束指令访存操作（不约束数据访存操作）。

4) Power 存储一致模型除了 FENCE 之外，提供了其他的顺序约束。对读操作一级高速缓存（L1），如果后续的访存操作需要用一级缓存的结果计算访存地址，且后续指令与该指令有依赖关系，则不能乱序执行。

2.2.4 几种商用处理器的存储器一致性模型的对比

一直以来，大家对存储一致模型的归类都是根据其两个重要的特征：(1) 对程序序的放松；(2) 对写原子性的放松[4][5]。

关于程序序，对来自同一处理器不同访存地址的操作，不同的存储一致模型对读后写、读后读、写后写、写后读具有不同的约束。

关于写原子性，不同的存储一致模型对写操作在全局可见之前，是否允许读操作提前得到该值（即：一个写操作产生的写无效消息或写更新消息到达所有高速缓存备份之前，读操作是否允许返回写操作的值）有不同的约束。

根据上述两个特征，我们用表 4 总结 SPARC、X86、POWER 存储一致模型。

表4. 不同的存储一致模型

存储一致模型	写→读	写→写	读→读	读→写	提前读取其它处理器写操作	提前读取本地写操作	正确性保证
RMO	√	√	√	√		√	MEMBAR 指令
PSO	√	√				√	STBAR 指令、读改写指令
TSO	√					√	读改写指令
X86	√					√	顺序指令、锁前缀指令
POWER	√	√	√	√	√	√	SYNC 指令

3 高速缓存同一性

3.1 高速缓存同一性的基本概念

在共享存储的多核处理器系统中，高速缓存结构可以将共享存储空间中的数据缓存在本地，加速各处理器获取数据的过程。由于每个处理器看到的存储器视图都是通过本地的高速缓存得到的，因此对于同一个存储位置的数据而言，不同的处理器可能会获取到不同的数据值。如图 1 所示[14]，三个带有高速缓存的处理器通过总线形式连接到主存，高速缓存及主存相关操作按照图中 1~5 的顺序执行。在处理器 P3 将其缓存中的 u 值由 5 改成 7 之后，该缓存若是直写式，则主存中相应位置的数据也会更新。但是，操作 4 读取依然是 u 的旧值 5，而非新值 7。

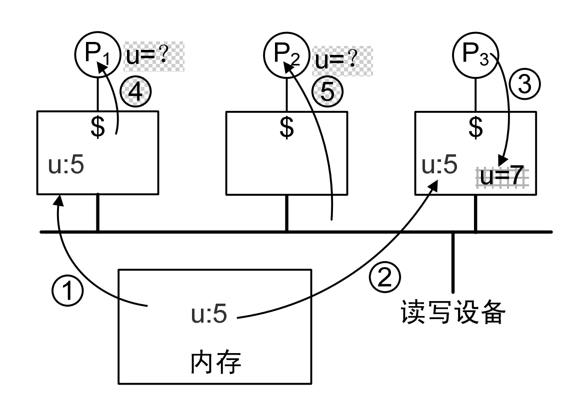


图1. 高速缓存同一性问题示意图

多核处理器中的高速缓存同一性问题是普遍存在的。在多核处理器设计过程中，必须引入缓存同一性维护机制，保证多个处理器对主存中同一位置的读取操作总能获得最新写入的数据值。这也是制约多核处理器性能提升的一大瓶颈。

实际上，高速缓存同一性协议的维护需要保证两点[15]。一是写操作的传播：某个处理器的写操作对于其他任意处理器都是可见的；二是写操作的串行化：不同处理器对同一存储位置的写操作，在所有处理器看来都是按照同样的顺序执行的，处理器在读取存储器时总能获取最新写入的值。

高速缓存同一性协议中广泛采用的有两类协议——基于目录的协议和基于侦听的协议[15]。基于目录的协议对于共享存储其中的数据块设置目录项来跟踪、记录其状态信息，从而知道哪些节点应该对请求做出何种操作。目录协议的实现方案较多，根据目录存储方法的不同可以分为集中式目录协议（如 Tang）、分布式目录协议（如 Censiert）等。集中式目录协议是指在主存中只用一个目录来标识数据在各处理器缓存中的存储情况。集中式目录也有几种实现方案，全映射目录方案就是一种典型的集中式目录协议。分布式目录协议又称链表式目录协议。这种协议是将目录分散到各处理器的缓存中，并用链表将有关的缓存连接在一起。每个主存中的数据块建立一个数据链表，每个链表只包含拥有该数据块副本的缓存。基于侦听的协议是另一种多核处理器中维护高速缓存同一性的常用方法。具体来说，缓存控制器通过共享总线事务来更新本地数据的一致性状态。这里，高速缓存同一性的维护可以看作是一组有限状态机的状态变迁。挂接在总线上的节点接收到同一性请求后，会根据消息类型和自身状态更新数据状态并向请求者做出响应。避免出现缓存内容不一致的方案有两种，分别为“写作废”与“写更新”。“写作废”是将所有远程拥有相同数据块副本的缓存中对应内容“作废”，使有效数据只有一个，典型的协议有 MSI 协议和 MESI 协议等；“写更新”是将存有相同数据块副本的缓存中对应内容“更新”，可能出现多个有效数据。典型的协议有 Dragon 等。不过，“写更新”方案需要将更新内容发送到所有须更新的缓存中，会大大增加总线的负担，一般使用得不多。

下面，我们重点介绍几种商业处理器中采用的高速缓存同一性相关技术。

3.2 商用多核处理器中的高速缓存同一性协议

3.2.1 Intel QPI

由于 FSB（front-side bus, 前端总线）的自身技术特点，英特尔公司意识到，虽然 FSB 频率已经较高，但是处理器与芯片组之间的性能瓶颈仍未改变。单纯通过提高处理器的外频和 FSB 带宽，很难带来明显的性能提升。2008 年后期，英特尔正式发布了替代 FSB 技术的新的互连设计方案——QPI（QuickPath Interconnect）互连技术[16]。

QPI 实际的官方名称是 Common System Interface （CSI），即公共系统界面，是一种包传输的串行式高速点对点连接技术，用来实现芯片之间的直接互连，而非通过 FSB 连接到北桥，并且，其数据传输速率相较于 FSB 有大幅提升。

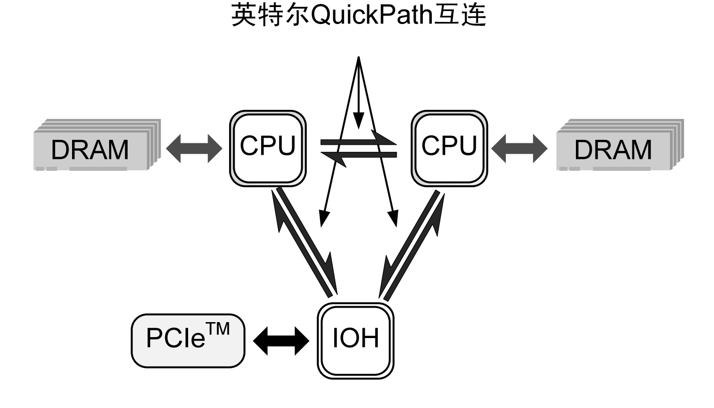


图2. Intel QPI 技术体系结构示意图

英特尔希望在多核处理器时代处于优势地位，首要的任务就是要解决系统资源的分配难题，充分发挥多核的优势，这也是英特尔推出 QPI 总线技术的最终目的。QPI 技术已推出了 1.0 和 1.1 版本。图 2 是 QPI 的体系结构示意图[17]。图中，QPI 不仅用于处理器与 IOH（IO Hub）的互连（内存控制器已集成于处理器中），同时也用于处理器之间的直接互连。图中的 IOH 现在也常集成到处理器内部，不再作为板级单独部件出现。

QPI 不仅应用于处理器之间的交互，在 Intel 处理器内部，QPI 也常用于最末级缓存（Last Level Cache, LLC）（通常是 L3）的缓存代理（CA, Caching Agent）和宿主代理（HA, Home Agent）之间的通信。如果目的 HA 不在本处理器中，则也需要通过处理器间的 QPI 交互。

QPI 的接口包括物理层、链路层、路由层、传输层和协议层等多个组成部分，详细的组成结构见图 3[16]。

其中，QPI 协议层制订了一系列规则，用于保证分布式共享存储系统的缓存同一性。常见的 QPI 缓存同一性协议分为 Home Snoop（宿主侦听）和 Source Snoop（数据源侦听）两种[16]。图 4 为 QPI 支持的缓存同一性协议的 Home Snoop 模式示意图。

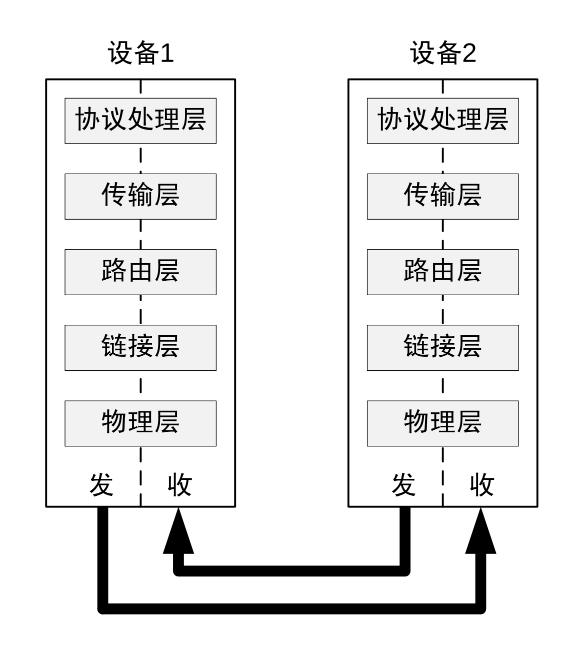


图3. QPI 接口示意图

在 QPI 1.0 版本中，Home Snoop 的具体执行步骤可以分为四步[16]：(1) 请求者向宿主节点发送请求，(2) 宿主利用自己的目录结构选择可能拥有主存中数据块副本的节点，发送侦听请求，(3) 接收到侦听请求的节点会给宿主节点一个反馈，并将数据直接传送给请求者，

(4) 宿主节点检查数据是否已经由其他节点传送给请求者，否则，将直接把数据传送给请求者，并告知整个 Home Snoop 过程结束。由于 Home Snoop 模式使用到了目录结构，准确的说，它是一种基于目录的高速缓存同一性协议。值得注意的是，在 QPI 1.1 版本中，Home Snoop 协议并不强制使用目录结构。

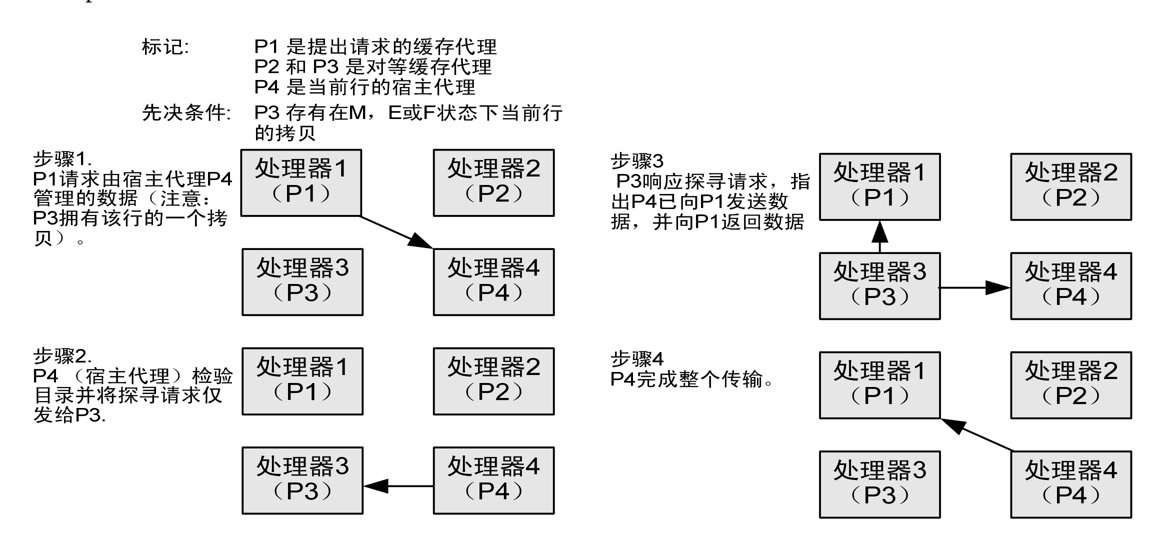


图4. QPI 高速缓存同一性协议的 Home Snoop 模式

图 5 为另一种模式 Source Snoop 的示意图。

在 QPI 1.0 版本中，Source Snoop 的具体执行步骤可以分为三步[16]：(1) 某个处理器向宿主节点发送请求，并同时向其他所有节点发送侦听要求，(2) 接收到侦听要求的节点会给宿主节点一个反馈，拥有主存中数据块副本的节点还会直接将数据传送给请求者，(3) 宿主节点告知请求者整个 Source Snoop 过程结束。

Home Snoop 模式一般用于获得更好的可伸缩性，常用于较大型的多处理器系统互连。通过 Home Snoop 模式，请求者可以通过请求宿主节点发送侦听要求，满足条件的节点才会收到侦听要求并回应，避免了 Source Snoop 模式中的直接式的回应，大大降低了互连中的流量和带宽压力。Source Snoop 模式则较适用于较小型的多处理器系统互连。互连中的节点较少时，使用 Source Snoop 模式与 Home Snoop 模式在互连流量方面并不会有很大的差距，但是前者由于不用等待宿主节点将侦听要求发送给其他节点，在接收回应的时延方面有很大优势。

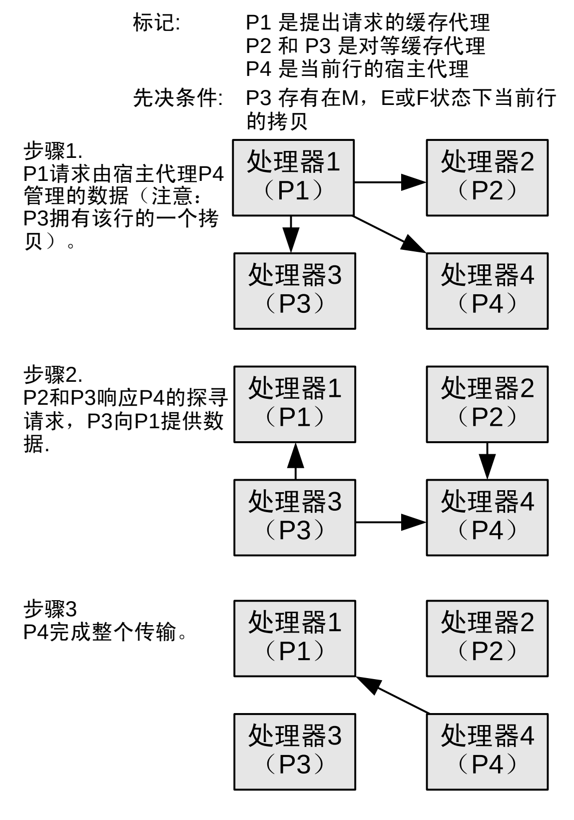


图5. QPI 高速缓存同一性协议的 Source Snoop 模式

QPI 采用的是基于经典的高速缓存同一性协议——MESI 协议开发的 MESIF 协议[18]，该

协议主要用于解决 CC-NUMA[4]处理器架构的缓存同一性问题。需要说明的是，本文提及的 CC-NUMA 结构，特指通过 Intel QPI 技术或者 AMD HT 技术直接互连构成的多处理机系统。

相对于 MESI 协议，MESIF 协议引入了一个 F（Forward，转发）状态，在 CC-NUMA 架构的处理器系统中，可能在多个处理器的缓存中存在相同的数据副本，其中只有一个 缓存行的状态为 F，其他缓存行的状态都为 S。当缓存行的状态位为 F 时，缓存中的数据与存储器一致。当一个数据请求方读取这个数据副本时，只有状态为 F 的缓存行可以将数据副本提供给数据请求方，而状态为 S 的缓存不能提供该数据。从而，MESIF 协议有效解决了 CC-NUMA 处理器结构中由于所有状态位为 S 的缓存同时提供数据副本给数据请求方而可能造成的网络拥塞。如图 6 所示，左边是 MESI 协议，右边是 MESIF 协议。图中两者都是由请求者向每个节点发送数据请求。其中，点虚线表示数据请求，点划线表示响应。

在 CC-NUMA 处理器系统中，如果状态为 F 的数据副本被其他处理器拷贝时，F 状态会被迁移，新建的数据副本的状态将为 F，而旧的数据副本的状态将改为 S。当状态为 F 的缓存行被改写后，CC-NUMA 处理器系统需要首先使状态为 S 的其他的缓存行无效，之后将该缓存行状态更新为 M。

MESIF 协议由 Intel Boxboro-EX 平台（由多个 Nehalem-EX 处理器组成）引入，目前英特尔没有公开其详细设计文档。MESIF 协议的详细实现与 QPI 的接口最高层——协议层相关。如前文所述， MESIF 协议主要解决 CC-NUMA 架构中 SMP[5]子系统之间的缓存同一性问题。而在独立的 SMP 处理器中，一般依然使用传统的 MESI 协议。

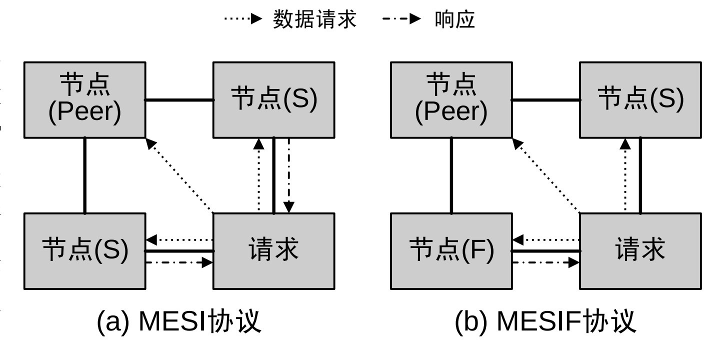


图6. QPI 高速缓存同一性协议的 Source Snoop 模式

3.2.2 AMD HT

HT 技术的全称为 HyperTransport，是 AMD 公司为 K8 平台专门设计的高速串行总线，如图 7 所示[19]。它的发展可以追溯到 1999 年，原名为 LDT (Lightning Data Transport)，即闪电数据传输。2001 年 7 月此项技术正式推出，AMD 同时将其更名为 HyperTransport，并推向产业界，组建 HT 技术开放联盟 HTC[6] 。

基础原理方面，HT 技术与 PCI-E（PCI Express）非常相似，都是采用点对点的单双工传输线路，引入抗干扰能力强的 LVDS 信号技术，命令信号、地址信号和数据信号共享一个数据路径，支持 DDR 双沿触发技术等。只是两者用途不同，PCI-E 是系统总线，而 HT 则是作为芯片之间的互连，互连对象可以是处理器与处理器、处理器与芯片组、芯片组的南北桥之间等，属于计算机系统的内部总线。

HT 技术发展历程主要经历了四个阶段，第一代 HT 技术 1.0 和 1.1 版本分别在 2001 年和 2002 年发布；2004 年 2 月，HT 技术联盟正式发布了 HT 2.0 版本；2006 年，AMD 正式发布 HT 3.0 规范；最新的 HT 3.1 规范由 AMD 组建的超传输技术联盟（HTC）在 2008 年 8

[4] cache-coherent non uniform memory access，一致性缓存非均匀内存寻址（流行的翻译为“连贯缓冲非统一内存寻址”）
[5] Symmetric MultiProcessing，对称多处理器架构
[6] HyperTransport Technology Consortium

月发布。

HT 的接口包括物理层、数据链路层、协议层、传输层和会话层，具体包括 CC-HT（Cache Coherent HT）技术和 nC-HT（non-Coherent HT）技术[19]。如图 8 所示，以使用 HT 技术的 AMD Opteron Shanghai 处理器为例，其采用的是 CC-HT 3.0 技术，提供最高 41.6GB/s 系统频宽[20]。多 Opteron 处理器也组成一个 CC-NUMA 架构，采用的是经典的 MESI 协议的另一个变种——MOESI 协议[15]。

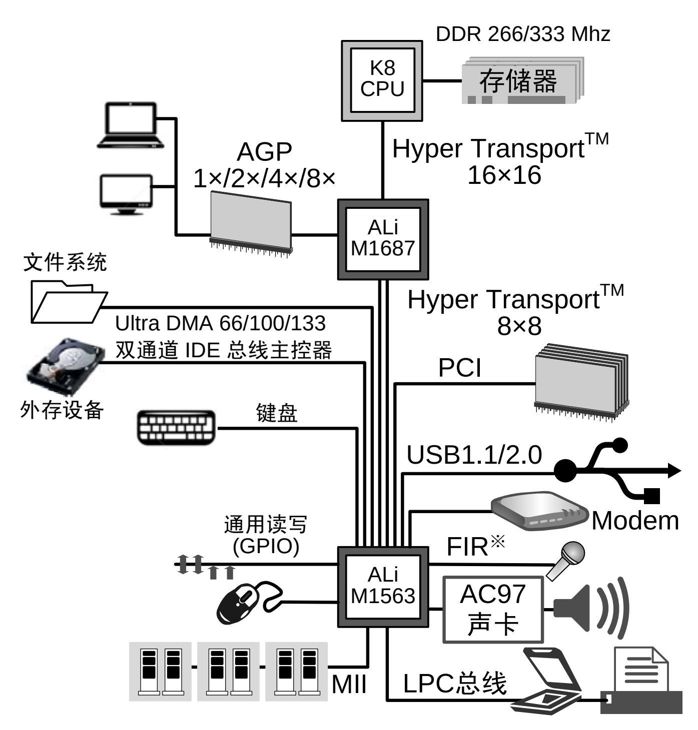


图7.　AMD K8 平台架构示意图

相对于前面提到的 MESIF 协议，MOESI 加入了另一个新的缓存行状态 O（Owned），并同时重新定义了 S 状态，M、E、I 状态与 MESI 协议中的各状态定义一致。图 9 是 MOESI 协议的高速缓存同一性模型示意图。其中，“Probe Read”表示主设备从其他处理器获取数据副本以读取数据的操作，而“Probe Write”表示主设备从其他处理器中获取数据副本以写入数据的操作；“Read Hit（读命中）”和“Write Hit（写命中）”表示主设备在本地缓存中获得了数据副本，“Read Miss（读未命中）”和“Write Miss（写未命中）”则表示未能获取副本；“Probe Read Hit”和“Probe Write Hit”表示主设备在其他处理器的缓存中获得了数据副本。前面提到的 O 状态只有在本地或远程读命中时，由 O 状态自身保持，或者远程读命中时，由 M 状态转化这两种情形下形成。

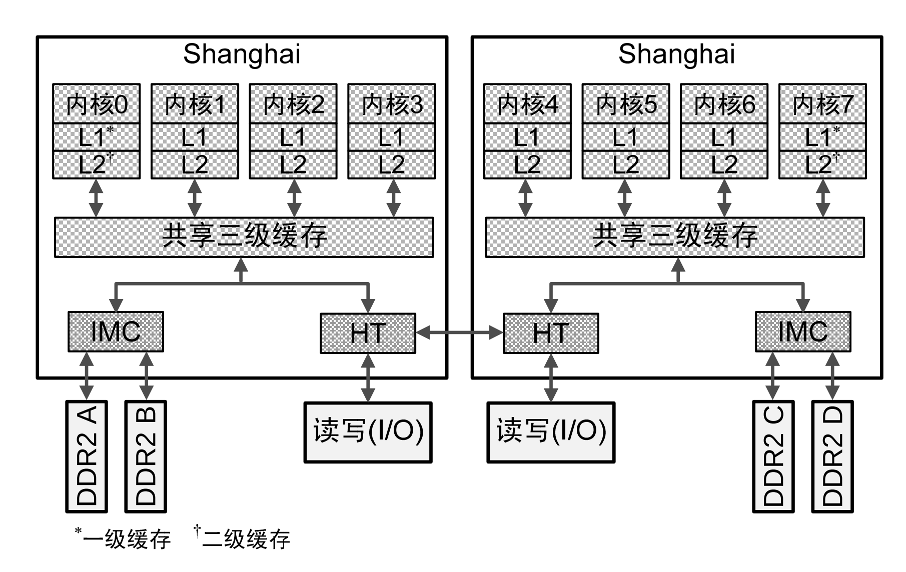


图8.　AMD Opteron Shanghai 处理器结构图

MOESI 模型中，状态为 O 时，表示当前缓存行中包含的数据是当前处理器系统最新的

数据拷贝，而且在其他处理器中一定具有该缓存行的副本，其他处理器的缓存行状态为 S。如果主存的数据在多个处理器的缓存中都具备副本时，有且仅有一个处理器的缓存行状态为 O，其他处理器的缓存行状态只能为 S。与 MESI 协议中的 S 状态不同，状态为 O 的缓存行中的数据与主存中的数据并不一致。

MOESI 模型中的 S 位状态的定义相对于 MESI 发生了一些细微变化。当一个缓存行状态为 S 时，其包含的数据并不一定与主存一致。如果在其他处理器的缓存中不存在状态为 O 的副本时，该缓存行中的数据与主存一致；如果在其他处理器的缓存中存在状态为 O 的副本，此缓存行中的数据与主存不一致。

在某些场合，使用 MOESI 协议将极大地提高缓存的利用率，因为该协议引入了 O 状态，从而在发送 Read Hit 的情况时，不必将状态为 M 的缓存回写到主存，而是直接从一个处理器的缓存将数据传递到另一个处理器。目前 MOESI 协议在实际应用中得到了广泛的应用，如 DEC Alpha，AMD x86，RMI RAZA 等系列处理器，ARM Cortex A5 和 SUN UltraSPARC 处理器也是用了这种协议。MOESI 协议不仅可用于 CC-NUMA 处理器架构，也可以用于优化 SMP 系统的缓存同一性。

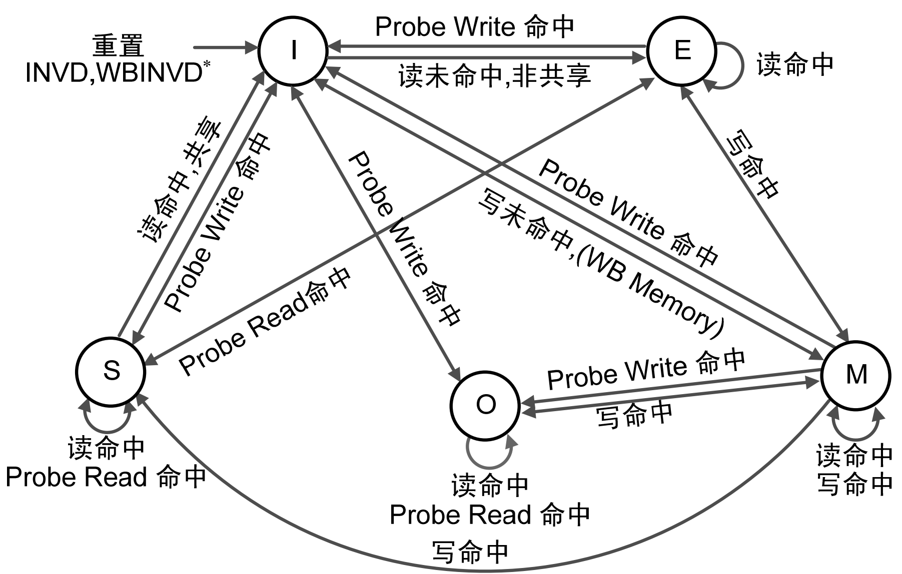


图9. MOESI 协议状态转化图

3.3 几种高速缓存同一性协议的对比

表 5 对 MESI、MOESI 和 MESIF 集中多处理器系统的高速缓存同一性协议各状态及其相互转化进行了比较。其中，干净副本指未经修改的副本，脏副本即一个内容被修改了的主存数据副本，需要更新到主存里面去。

AMD 的基于 CC-HT 技术的 Opteron 处理器使用的是 MOSEI 协议，英特尔的 Xeon 处理器使用的是 MESI 协议，而其之后推出的 Nehalem 处理器使用的是基于 QPI 技术的 MESIF 协议。

MESI 协议作为一种经典的“写无效”式的高速缓存同一性协议，也是由早期的 MSI 协议发展而来，因为逻辑简单，总线传输量较少，被广泛使用。但主存中的数据若在多个缓存中存在副本，当请求者发出对该数据的请求时，将会引起多个含有其副本的处理器响应，在系统规模较大时，会大大增加系统开销。

干净副本的共享较易处理，关键是对脏副本的处理。显然，多个副本中同一时间内只能

有一个可以被写入主存。MESIF 协议中，只具有一个副本的 E 状态在被写入的时候只需要简单地转化为 M 状态；如存在 F 状态副本，其被写入时则会导致其所有的 S 状态副本都被置为无效；由于 S 状态副本不允许转发，也不允许被写入，这些副本所在的处理器要再次使用这个副本时，需要再次向原始 F 状态副本请求，F 状态副本现在已经转化为 M 状态副本，被请求状态下 M 状态副本会写入主存并重新转化为 F 状态，不被请求时则可以保持在 M 状态，并可以不必实时写入主存以降低对主存带宽的占用。另一方面，由于 MESIF 协议只允许多个共享副本当中的 F 状态副本被写入主存，在多个处理器均需要写入一个缓存行的时候，会出现 F 状态副本在各个处理器之间不停传输的“弹跳”现象，和令牌环相似。这样会降低系统性能，特别是 F 状态副本不在其所在的原始主存空间的时候。

Opteron 处理器使用的 MOSEI 协议不需要 M 状态副本写入主存就可以进行共享，此时这时 M 状态会转变为 O 状态，共享后的脏副本被标记为 S 状态。这避免了一次写入主存的操作，节约了一些系统开销；当再次写入 O 状态副本时，其他的状态 S 副本同样会被设置为无效。MOSEI 协议也只允许多个共享副本当中的 O 状态副本被写入，同样也存在着 MESIF 协议中的“弹跳”现象。

表5. MESI、MOESI 和 MESIF 协议比较

高速缓存同一性协议						
缓存行状态	干净/脏	唯一	可写	转发	可安静地转化成的状态	说明
MESI over FSB(Intel Xeon 处理器)						
M(Modified)	脏	是	是	是		被请求时需先写入主存
E(Exclusive)	干净	是	是	是	M、S、I	被写入时转化为 M 状态
S(Shared)	干净	否	否	是	I	可以提供数据
I(Invalid)	-	-	-	-		不能读取
MOESI over HT(AMD Opteron 处理器)						
M(Modified)	脏	是	是	是	O	被请求时不需要写入主存，而仅仅转化为 O 状态
O(Owner)	脏	是	是	是		主副本转换为其他状态需要先写入主存
E(Exclusive)	干净	是	是	是	M、S、I	被写入时转化为 M 状态
S(Shared)	干净或脏	否	否	否	I	可以同时为干净或者脏，主副本被写入时转为无效
I(Invalid)	-	-	-	-	-	不能读取
MESIF over QPI (Intel Nehalem 处理器)						
M(Modified)	脏	是	是	是		被请求时需要先写入主存
E(Exclusive)	干净	是	是	是	M、S、I、F	被写入时转化为 M 状态
S(Shared)	干净	否	否	否	I	主副本被写入时转为无效
I(Invalid)	-	-	-	-	-	不能读取
F(Forward)	干净	是	否	是	S、I	主副本被写入时转换为 M 状态，并使其他 S 副本无效

4 总结

存储器一致性和高速缓存同一性问题一直是共享存储的多处理机系统中的技术难点。特别是在当前，片上多核技术已成为处理器设计主流的背景下，这一问题越来越多地受到各大处理器设计生产商的关注和重视。本文主要对存储器一致性和高速缓存同一性问题进行了简要描述，同时针对诸多商用处理器中相关技术的近期发展状况进行了分类介绍，并针对其中的具体方案进行了对比分析。作为片上多核设计中的重要一环，存储器一致性和高速缓存同一性技术的发展需要学术界和工业界的共同促进。

参考文献：

[1] Kunle Olukotun, Basem A.Nayfeh, et al., The Case for a Single-Chip Multiprocessor, Architectural Support for Programming languages and Operating Systems. Proceedings of 7th ASPLOS, Cambridge, MA, 1996.10

[2] Kai Hwang, Zhiwei Xu. Scalable Parallel Computing: Technology, Architecture, Programming. The McGraw-Hill Companies, Inc. NY. 1998

[3] 张晨曦等，*计算机体系结构*，高等教育出版社。2000.5

[4] S. V. Adve and K. Gharachorloo, Shared memory consistency models: A Tutorial, IEEE Computer, 66-76, December 1996.

[5] Arvind and J. –W. Maessen. Memory model = instruction reordering + store atomicity. In proceedings of the 33nd Annual International Symposium on Computer Architecture, June 2006.

[6] S. Owens, S. Sarker, and P. Sewell. A better X86 memory modes: X86-TSO。 In proceedings of the conference on Theorem proving in the higher order logics, 2009.

[7] S. Sarker, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridger, T. Brabant, M. O. Myreen, and J. Alglave. The semantics of X86-CC multiprocessor machine codes. In proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 379-391, 2009

[8] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Communications of the ACM, July, 2010.

[9] Intel 64 and IA-32 architectures software developer's manual. Intel Corporation, 2001, Feb. 2008, Mar. 2010.

[10] IBM. Power ISA Version 2.06 Revision B. http://www.power.org/resources/downloads/ PowerISA_V2.06B_V2_PUBLIC.pdf, July 2010.

[11] IBM. Book E: Enhanced PowerPC Architecture, version 0.91, July 21, 2001.

[12] Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. Morgan & Claypool Publishers, 2001.

[13] SPARC International, Inc. The SPARC Architecture Manual Version 9. http://www.sparc.org/standards/ SPARCV9.pdf, 1994.

[14] David E. Culler, Jaswinder Pal Singh, et al.. Parallel Computer Architecture: A Hardware/Software Approach, Second Edition. 2003

[15] 黄安文与张民选，多核处理器 cache 一致性协议关键技术研究。*计算机工程与科学*，2009。31：第 104-108 页。

[16] An Introduction to the Intel QuickPath Interconnect. Intel Corporation. http://www.intel.cn/content/dam/doc/ white-paper/quick-path-interconnect-introduction-paper.pdf, 2009.1

[17] Ziakas D., Baum A., et al. Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. The proceedings of 18th IEEE Symposium on High Performance Interconnects. 2010: 1-6.

[18] Binfeng Qian, Limin Yan. The Research of the Inclusive Cache used in Multi-Core processor. Proceedings of the 2008 International Conference on Electronic Packaging Technology & High Density Packaging. 2008:1-4.

[19] HyperTransport Technology Overview. AMD Corporation. http://www.hypertransport.org/docs/uploads/ HT_General_Overview.pdf, 2011.

[20] Danial Hackenberg, Daniel Molka, et al. Comparing Cache Architectures and Coherency Protocols on

X86-64 Multicore SMP Systems. Proceedings of the MICRO’09. New York, NY. December 12-16, 2009.

作者简介：
周　君： 计算机体系结构国家重点实验室，博士研究生，zhoujun@ict.ac.cn
唐士斌： 计算机体系结构国家重点实验室，博士研究生